Playing by the Rules: Rewriting as a practical optimisation technique in GHC

Simon Peyton Jones [a,1] Andrew Tolmach [b,2,3] Tony Hoare [a,4]

[a] *Microsoft Research Ltd, St George House, 1 Guildhall St, Cambridge CB2 3NH, England*

[b] *Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, USA*

Abstract

We describe a facility for improving optimization of Haskell programs using rewrite rules. Library authors can use rules to express domain-specific optimizations that the compiler cannot discover for itself. The compiler can also generate rules internally to propagate information obtained from automated analyses. The rewrite mechanism is fully implemented in the released Glasgow Haskell Compiler.

Our system is very simple, but can be effective in optimizing real programs. We describe two practical applications involving short-cut deforestation, for lists and for rose trees, and document substantial performance improvements on a range of programs.

1 Introduction

Optimising compilers perform program transformations that improve the efficiency of the program. However, a compiler can only use relatively shallow reasoning to guarantee the correctness of its optimisations. In contrast, the programmer has much deeper information about the program and its intended behaviour. For example, a programmer may know that

```
integerToInt (intToInteger x) = x
```

(where `Integer` is the type of infinite-precision integers, and `Int` is 32-bit integers), but the compiler has little chance of working this out for itself. While programmers are unlikely to write such expressions themselves, they can

[1] Email: `simonpj@microsoft.com`

[2] Work performed in part while visiting Microsoft Research Ltd.

[3] Email: `apt@cs.pdx.edu`

[4] Email: `thoare@microsoft.com`

easily appear when aggressive inlining brings together code that was written separately.

In this paper we explore a very simple idea: *allow the programmer to specify program properties that the compiler can use to improve performance, by treating each property as a rewrite rule*. In effect, we give the programmer the ability to extend the compiler with domain-specific optimisations, giving it specialised knowledge about the particular vocabulary of functions that are used heavily in a particular program. Our setting is that of the purely functional language Haskell, because the lack of side effects makes it possible to state many properties simply, without complex side conditions, and to exploit them using only local information.

We make the following contributions:

- We describe a concrete design, which is fully implemented in the released Glasgow Haskell Compiler, an optimising compiler for Haskell (Section 2; Section 4).
- We describe two practical applications of the technique, one to perform list fusion in the Haskell standard Prelude (Section 3) and other to perform tree fusion in an application-specific library (Section 6).
- We show that rewrite rules can also be generated automatically as a result of compiler analyses, and then constitute a useful way to exploit specialised versions of functions (Section 5).

The idea of allowing the programmer to specify domain-specific compiler extensions is not new (Section 7), but it has not yet been widely successful. Our principal selling point is *simplicity*. Rewrite rules are expressed declaratively using the syntax of Haskell itself, and not in a separate meta-language. They use very simple first-order pattern matching, have no side conditions, and are applied using a trivial strategy. Yet they are effective in real programs, assuming some cooperation from library writers.

We currently make no attempt to verify that programmer-specified rules are consistent with the underlying function definitions that they purport to describe. Rather, a programmer who adds a rule implicitly incurs a proof obligation, in much the same way as a user of GHC's `unsafePerformIO`. In addition, a rule's effect on program performance can be tricky to predict. For these reasons, the rules mechanism in its present form is primarily intended for use by "expert" programmers and library authors, who understand GHC's optimization behavior.

Having the rules explicitly codified does, however, raise the possibility of feeding the same program into a theorem prover, and having it prove that the rules are consistent with the implementation, perhaps with some human assistance — although we have not explored this avenue so far.

Adding explicit equational properties to programs has already been advocated for other purposes. They can serve to document the intended behavior of the program, independently of the implementation, and have been used to explore efficient algorithms and as a design methodology that reduces the incidence of programming error [4]. Another advantage may be reaped in testing and debugging of programs, where they can play the role of a test oracle [8]. Perhaps the additional incentive of efficiency gains in compilation will help convince the world that equational specification is a worthwhile part of the programming process.

2 The basic idea

Consider the familiar `map` function that applies a function to each element of a list. Written in Haskell, `map` looks like this:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Now suppose that the compiler encounters the following call of `map`:

```
map f (map g xs)
```

We know that this expression is equivalent to

```
map (f . g) xs
```

(where "`.`" is function composition), and we know that the latter expression is more efficient than the former because there is no intermediate list. But the compiler has no such knowledge.

One possible rejoinder is that the compiler should be smarter — but the expert programmer will always know things that the compiler cannot figure out. Another suggestion is this: allow the programmer to communicate such knowledge directly to the compiler. That is the direction we explore here.

The Glasgow Haskell Compiler (GHC) allows the programmer to add a *rule* to the program thus:

```
{-# RULES
  "map/map" forall f g xs.
            map f (map g xs) = map (f . g) xs
#-}
```

The "`{-# ... #-}`" brackets enclose a *pragma*, which is ignored by a non-optimising compiler. The `RULES` keyword identifies the pragma as defining a rewrite rule. The `"map/map"` part is an arbitrary string that names the rule; this name is used when reporting which rules fired during a compilation run in diagnostic mode. The body of the rule expresses the identity that

```
map f (map g xs) = map (f . g) xs
```

while the `forall` part identifies which of the variables in the rule body are universally quantified (`f`, `g`, and `xs` in this case), and which are constants bound elsewhere (`map` in this case).

The general form of a programmer-specified rule is

"*name*" `forall` $(v_1$`::`$t_1)$ ... $(v_m$`::`$t_m)$`.`$f\ e_1\ \ldots\ e_n$ `=` e

for $m, n \geq 0$. Here *name* is a string identifying the rule, as described above. The v_i are bound variables with associated types t_i; the types can be omitted unless required to make the rule type-check, and the entire `forall` clause can be omitted if there are no bound variables. f is an *unquantified* function or constant identifier (i.e., not one of the `forall`'d variables), and the e_i are arbitrary Haskell expressions. A `RULES` pragma can occur only at the top level of the program, and all the free variables of the rule, on both sides of the equation, must be in scope.

One can regard the rules for a function as extra (redundant) equations defining the function, thus:

```
map f []          = []
map f (x:xs)      = f x : map f xs
map f (map g xs) = map (f . g) xs
```

Unlike ordinary defining equations, of course, rules are not restricted to having constructors in the patterns on the left hand side.

Rewrite rules express identities that the programmer knows to be true, but GHC also assumes that they are *oriented*, so that the right hand side is preferable to the left. Throughout compilation, GHC tries to spot instances of the left hand side of a rule, and rewrite that call to the right hand side.

2.1 Assumptions

The ability to add rewrite rules to a program is a pretty powerful weapon, and raises a host of issues. In particular:

- GHC makes no attempt to verify that the rule is consistent with the underlying function definitions, apart from ensuring that the left and right hand sides of the rule have the same type. The whole point is that the rule asserts something that GHC is not smart enough to work out for itself! Moreover, if rule and implementation disagree, the implementation is just as likely to be wrong as the rule, perhaps even more so.

  Indeed, we might not even want the rule to be "true" in a concrete sense! For example, consider an abstract data type for sets. It is sound to give a rule expressing the fact that `union` on sets is commutative. But suppose our implementation represents a set by an unordered list. Then the concrete representation of `(a ‘union‘ b)` may differ from `(b ‘union‘ a)`, even though they represent the same sets.
- GHC makes no attempt to ensure that the right hand side is more "efficient" than the left hand side. One might like to say "simply write down some true properties, and the compiler will use them to optimise the program", but that is well beyond what we offer. Instead, as we discuss in Section 4, we

rely on the (fallible) programmer to specify oriented rewrite rules, and even a simple rewrite strategy. Using rules effectively therefore requires some understanding of how GHC works.

- GHC makes no attempt to ensure that the set of rules is confluent, or even terminating. For example, the following rule will send GHC into an infinite loop if it encounters a call to `foo`.

```
{-# RULES
    "commute"  forall x y. foo x y = foo y x
#-}
```

  There is a considerable literature on proving the confluence or termination of sets of rewrite rules; in particular, commutativity and associativity have received special study [3]. However, for us matters are seriously complicated by the other automatic rewrites that the compiler performs (beta reduction, inlining, case switching, let-floating, etc. [32]), so we are not able to take direct advantage of this work.

  For an optimising compiler, confluence seems too strong, since that would implausibly suggest a canonical optimised form for a program. Termination is certainly important, but has not proved to be a problem in practice.

2.2 Restrictions

As noted above, the pattern on the left hand side of a rule must be a function application (for some fixed function) or a constant. Here, for example, is a plausible rule that we cannot write:

```
{-# RULES
  "let/let"  forall x y e1 e2 e3.   -- ILLEGAL!
       let { x = let { y = e1 } in e2 } in e3
       = let { y = e1 } in let { x = e2 } in e3
#-}
```

The rule is illegal because the left hand side is not a function application. This restriction has two advantages. First, it underpins the idea introduced above, that a rewrite rule is simply an extra (redundant) equation defining a function. Second, it makes rule matching much more efficient, because the rules can be indexed by the function on the left hand side. At each call of `f`, GHC need only check matches for rules for `f`. If the left hand side of a rule could instead be an arbitrary expression, matching would likely be much less efficient.

The function-application restriction does mean that rules cannot be used to replace many of GHC's built-in transformations. Inlining, let-floating, beta reduction, case swapping, case elimination, and so on are all too complex to explain using our restricted language of rules. There are, however, some compiler transformations – such as specialisation – for which rules do prove directly useful, as we discuss in Section 5.

2.3 Library writers and library clients

Reading these assumptions and restrictions, one might reasonably ask: are rewrite rules going to be of practical use? It is certainly easy to shoot oneself in the foot.

For this reason, we regard a set of rewrite rules as something much more like *a domain-specific compiler extension* than a general programming paradigm. We expect rewrite rules to be written mainly by the author of a library. Such authors often go to great lengths to craft efficient data structures and algorithms. Rewrite rules give them the ability to explain deep truths about their code to the compiler, and thereby extend its ability to optimise client programs. We assume also a willingness to cooperate in the optimisation, to the extent of adapting library code to take advantage of the optimisation rules, as well as the other way round. In return, we hope to preserve a level of simplicity, in which the correctness of the optimisation rules (but not their effectiveness, unfortunately) is as easy to establish as that of all the other clauses in a declarative program.

In GHC the rewrite rules defined in a module are embedded in the compiler-readable meta-data (its "`.hi` file") that accompanies the module's object code. The client of the library never sees the rules, but GHC can nevertheless use them to optimise compositions of calls to functions supplied by the library. Rules are not explicitly exported or imported. Instead, when compiling module `M`, GHC can "see" all the rules given in any module imported by `M`, or in any module imported by these imports, and so on transitively. (Haskell's `instance` declarations have exactly the same property.)

A rule is *not* required to be in the same module as the function whose definition it extends. For example the `"map/map"` rule does not have to be given in the module that defined `map`. So rules can incrementally extend a function's definition. This is important, because a rule may describe the interaction of an imported function with one defined locally. Rules can also be given for a class member function, in which case they work on the corresponding function in each class instance.

Rewrite rules make perfect sense even if the library is written in another language, in which case the rules express facts about the foreign library. For example, in Reid's graphics library for Haskell he provides a whole section of the user manual devoted to algebraic optimisation laws that are satisfied by the library interface [33].

3 Rules in practice

In the rest of the paper we report on our experience of applying rewrite rules in practice. We have found two main classes of applications:

- Programmer-written rules in library code. This was our initial motivation, and we have used it to achieve list fusion (this section) and more ambitious tree fusion (Section 6).

- Automatically-generated rules, derived from some kind of program analysis, invisibly to the programmer (Section 5). This was an unexpected, but very persuasive, practical benefit of implementing the rewrite-rule technology.

3.1 Short-cut Deforestation

Our initial motivating example for adding rewrite rules was the case of list fusion. In earlier work, Gill, Launchbury, and Peyton Jones described so-called *short-cut deforestation*, a technique for eliminating intermediate lists from programs [16]. At the centre of the method is the single rewrite rule `"foldr/build"`:

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr k z [] = z
foldr k z (x:xs) = k x (foldr k z xs)

build :: (forall b. (a->b->b) -> b -> b) -> [a]
build g = g (:) []

{-# RULES
"foldr/build"
  forall k z (g::forall b.(a->b->b) -> b -> b) .
  foldr k z (build g) = g k z
#-}
```

The definition of `foldr` is conventional. The function `build` takes a “list” `g`, functionally abstracted over its cons and nil constructors, and applies `g` to the ordinary list constructors `(:)` and `[]` to return an ordinary list. (`g`’s type is a rank-2 polymorphic type, as discussed in [16]; we must specify it explicitly in order to make the rule type-check.) The rule states that when `foldr` consumes the result of a call to `build`, one can eliminate the intermediate list by applying `g` directly to `k` and `z`.

To give an example of applying this rule we must write list-consuming and producing functions using `foldr` and `build` respectively. For example:

```
-- (sum [5,4,3,2,1]) = 15
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

-- (down 5) = [5,4,3,2,1]
down :: Int -> [Int]
down v = build (\c n -> down' v c n)

down' 0 cons nil = nil
down' v cons nil = cons v (down' (v-1) cons nil)
```

Again, the definition of sum in terms of foldr is conventional. The function down returns a list of integers, from its argument down to 1. We express it as a call to build, using an auxiliary function down' which is abstracted over the functions it uses to construct its result. (We have called these functions cons and nil for old times' sake, but they are simply the formal parameters to down' and their names are insignificant.) It is somewhat inconvenient to write sum and down in this way, but that is the task of the author of the List library.

Now we can try fusion on the call (sum (down 5)):

```
sum (down 5)
=       {inline sum and down}
  foldr (+) 0 (build (down' 5))
=        {apply the foldr/build rule}
  down' 5 (+) 0
```

The intermediate list has been eliminated; instead down' does the arithmetic directly.

3.2 A real (albeit small) example

List fusion works well when the programmer does "bulk" operations over lists, and then it can be stunningly effective. Here is an example taken verbatim from the paraffins code [29], a small program that computes a list of all the hydrocarbon paraffins of a given size:

```
three_partitions :: Int -> [(Int,Int,Int)]
three_partitions m
  = [ (i,j,k) | i <- [0..(m `div` 3)],
                j <- [i..(m-i `div` 2)],
                let k = m - (i+j)
    ]

-- A test harness
main = print (length (three_partitions 4000))
```

The form [0..n] is Haskell's notation for the list of integers between 0 and n. The list comprehension builds the list of all triples (i,j,k) where i is drawn from the list [0..(m `div` 3)], and j is drawn from a similar list, and k is computed directly from i and j. Finally, the test harness prints the length applying three_partitions to 4000.

GHC translates range notation, [0..n], into an application of build, much as we did for down above. It translates a list comprehension into a build, using foldr to consume the sub-lists. Finally, the Prelude library function length is implemented using a foldr.

So in this program, all the intermediate lists are removed, leading to a dramatic drop in allocation. Without fusion, this program allocates 188 Mbytes;

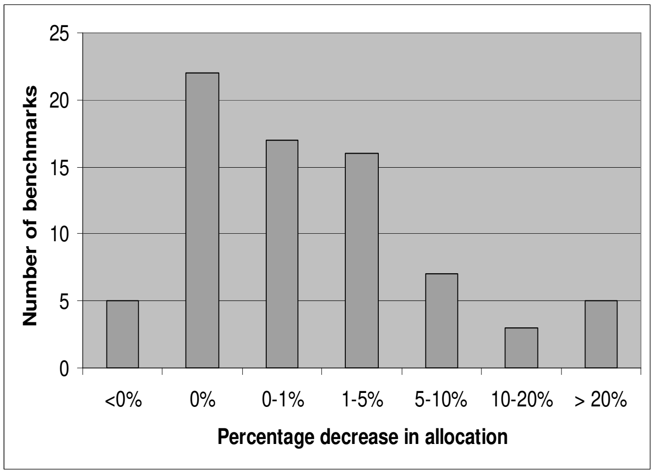

Fig. 1. Distribution of fusion effects on programs in "real" and "spectral" divisions of `nofib` benchmark suite, under ghc4.08.2.

when fusion is enabled, it allocates only 16 Mbytes. (Most of the allocation for the fused version is used for the stack, because the length computation is not properly tail-recursive, so the stack grows 1.3M activation records.)

3.3 Benchmark Results

Over a broader range of programs from the `nofib` benchmark set [29] the effect of enabling list fusion is very patchy, as Figure 1 shows. Fusion has no measurable effect on most programs but it gives a useful 5-25% reduction in allocation for a few. Only a very few programs are made worse, and the worst of these by less than 4%. One program, a parser called `parstof`, shows a 96% reduction; this turns to be because fusion transforms the (artificial) outer loop of the benchmark, causing the sample text input to be parsed once instead of 40 times!

The geometric mean improvement, about 5% if we omit `parstof`, seems disappointingly low, but we are undismayed. Compiler optimisations are like therapeutic drugs. Some, like antibiotics, are effective on many programs; such optimisations tend to be built into a compiler. Others are targeted at particular "diseases", on which they are devastatingly effective, but have no effect at all on most other programs. The rules mechanism allows library authors to add targeted, domain-specific optimizations without modifying the internals of the compiler.

We also hope that programmers may adopt a more modular programming style if they expect fusion to take place. For example, it is clearer to write

```
concat (map f xs)
```

than it is to write

```
foldr ((++) . f) [] xs
```

Yet programmers will sometimes write the latter form because it does not build an intermediate list. Section 6 gives an extended example of the way in which fusion can make modular programming practically efficient.

Finally, note that our measurements relate to un-modified benchmark programs. None of the functions in these programs use `build`, so fusion only occurs for compositions of functions from the Standard Prelude, whose functions we re-implemented using `foldr` and `build`. If the compiler were to transform user-written functions to use `foldr` and `build` we might see greater benefits — but that is beyond the scope of this paper, and in any case certainly would require compiler modification [25].

4 From theory to practice: the sticky details

So far we have implied that one simply needs to add one rewrite rule, and re-implement some key functions using `foldr` and `build`. In practice, though, we encountered a number of obstacles, which we discuss in this section, after first explaining our implementation of rewriting.

4.1 Implementation

The implementation of the rule rewriting mechanism within GHC is largely straightforward. The front-end has been extended to handle rule parsing, type checking, and translation into the `Core` intermediate language. The GHC optimiser is structured as a number of separate passes over `Core` expressions [32,31]. The most fundamental pass – iterated many times – is the *simplifier*, which performs inlining, case simplification, and eta-expansion in the course of a single top-to-bottom traversal of the program. To support rewriting, we just modified the simplifier to check each function application it encounters against a list of active rules; if the application matches the rule LHS pattern, it is replaced by a suitably instantiated version of the RHS. Matching is performed modulo eta-reduction, so that, e.g., an application of `\x -> f x` matches a rule with head `f`. We need to take a little care to make sure that the rule remains attached to the right function if alpha-renaming takes place.

Including rules adds a modest overhead to GHC compilation time. For example, using the list fusion rules described in Section 3 increases compilation times an average of 5% over the `nofib` benchmark suite. Some of this increase is probably due to performing conventional optimisations that are enabled by rule-based rewrites. In any case, we have made no serious attempt to analyse or optimise this aspect of compiler performance, so it can probably be sped up should this prove important.

4.2 Phases

The first obstacle faced when defining rules is a subtle interaction between function inlining — a transformation that GHC does aggressively [31] — and rule application. Returning to our `sum/down` example, we can see:

- `sum` and `down` must both be inlined before the rule can fire.
- On the other hand `foldr` and `build` must *not* be inlined. For example, inlining `build` before firing the rule would give

  ```
  foldr (+) 0 (down' 5 (:) [])
  ```

  and we have lost the fusion opportunity.
- However, once we have run out of opportunities to use the `foldr/build` rule, we *should* inline `build`. Recall that its definition is both small and higher-order:

  ```
  build g = g (:) []
  ```

  Inlining a function like this is very beneficial (`g` is often an explicit lambda).

These considerations led us initially to the following two-phase strategy:

(i) "Black list" any function that appears on the left hand side of a rule. Run the simplifier, applying rewrite rules, but refraining from inlining any black-listed functions.

(ii) Empty the black list, and repeat the exercise, so that previously-blacklisted functions will now be inlined.

Alas, two phases are not necessarily enough. In general, a program uses many layers of abstract data types, each implemented using the layer below. First we want to apply rewrite rules for the top-level ADT; then we want to expose its implementation (only to the compiler, of course) by inlining, and apply rewrite rules for the next layer; then we want to inline that layer and apply rewrite rules for the layer below; and so on.

Organising rules into phases is a form of *rewriting strategy*, a subject that has received considerable attention [36,22,9,37] However, one of the merits of rewrite rules is their simple, declarative nature: "here is a true fact: please use it whenever possible". We resist polluting this story with elaborate rewrite strategies. Nevertheless, it seems that some very simple strategy, such as a phase organisation is necessary. To gain experience, we have implemented the following very simple scheme.

The compiler runs the simplifier repeatedly, each run having a smaller [5] *phase number* than the previous one. A function may have an *inline pragma*, and this pragma can tell the compiler which phase to inline the function in. For example:

[5] Our present implementation uses increasing phase numbers, but we plan to reverse this shortly.

```
{-# INLINE 1 build #-}
build g = g (:) []
```

means "inline `build` in phase 1 (or smaller), *even if it appears on the left-hand-side of a rule*".

This scheme is clearly very crude. It requires the programmer to know something about GHC's phases, which is undesirable; and assigning phase numbers is not modular, requiring a global view of the program. Another question is whether it is better to annotate `INLINE` pragmas or the rules themselves. Various more elaborate schemes have occurred to us — using the module hierarchy, for example — but we have taken the view that we should refine the scheme in the light of practical experience, rather than implement an over-elaborate scheme right away.

4.3 Backing out

Suppose fusion does not take place. That is, suppose we have an isolated call `(down 34)`. It would be bad to actually implement `down` using `build` and `down'`, because doing so involves much more run-time function-passing than a straightforward implementation of `down`. It is unacceptable for programs to run slower in the (common) places when fusion fails than using the original library.

One solution is to rewrite `down'` to be non-recursive, and inline vigorously:

```
down :: Int -> [Int]
down v = build (\c n -> down' v c n)

down' v cons nil = go v
               where
                 go 0 = nil
                 go v = cons v (go (v-1))
```

Now suppose we have inlined `down` at a call `(down 34)`, but alas it has not fused with a `foldr`. We can now inline as follows:

```
build (\c n -> down' 34 c n)  -- Did not fuse
=          { Inline build }
  down' 34 (:) []
=          { Inline down' }
  (go 34) where
            go 0 = []
            go v = v : go (v-1)
```

This code is as good as the original, straightforward implementation of `down` — because it *is* the original, straightforward implementation of `down`! The trouble is that we have effectively made a complete copy of the straightforward code at every call site. While this is acceptable for a function as small as `down`, it would be quite undesirable for larger functions.

An alternative solution, and the one we generally adopt, is to have the library author add a new definition and rewrite rule:

```
downList :: Int -> [Int]
downList 0 = []
downList v = v : downList (v-1)

{-# RULES "downList"
    forall v. down' v (:) [] = downList v #-}
```

An isolated call to `(down 34)` would now transform as follows:

```
down 34
=          {Inline down}
  build (down' 34)
=          {Inline build}
  down' 34 (:) []
=          {Apply "downList" rule}
  downList 34
```

The `"downList"` rule spots the special case in which `down'` is applied the standard list constructors, and transforms the call to use the directly-coded `downList` function.

4.4 One-shot lambdas

Here is the definition of `map` in terms of `foldr` and `build`:

```
map f xs = build (\c n -> foldr (c . f) n xs)
```

Now, suppose we find an application `(map f (build g))`. We want to transform the call like this:

```
map f (build g)
=        {Inline map}       DANGER!
  build (\c n -> foldr (c . f) n (build g))
=        {Apply foldr/build rule}
  build (\c n -> g (c . f) n)
```

The difficulty is in the step marked `DANGER!`. Here we substitute `(build g)` for `xs` in the body of `map`, but this occurrence of `xs` is under a lambda abstraction. In general, one can make a program run arbitrarily more slowly by substituting a redex inside a lambda abstraction, so GHC usually does something more conservative:

```
map f (build g)
=        {Inline map}       SAFE!
  let xs = build g
  in build (\c n -> foldr (c . f) n xs)
```

Alas now the `foldr/build` rule cannot fire!

The solution is to observe that the abstraction `(\c n -> ...)` is a *one-shot lambda*; that is, it is a function that is only called once. Why? Because it is the argument to `build`, and `build` simply calls its argument, passing `(:)` and `[]`. Substituting inside one-shot lambdas is perfectly safe.

The Right Thing To Do is to analyse the program for one-shot lambdas and act accordingly. A type-based analysis that achieves this (among other things) is described by Wansbrough [38], but it is not yet fully implemented in GHC. Instead we have a temporary hack that spots the special case of an application of `build`.

4.5 Sharing

Consider this function

```
f x = sum (filter (> x) [1..10])
```

One might expect all intermediate lists to be eliminated from this function, but GHC correctly spots that the expression `[1..10]` can be floated out:

```
one_to_ten = [1..10]
f x = sum (filter (> x) one_to_ten)
```

Alas, now the filter consumer cannot fuse with the `[1..10]` producer. Floating out `one_to_ten` would be a good transformation if the producer — in this case `[1..10]` — were more expensive. It would be worth losing the fusion, in order to share the computation of `one_to_ten` among all calls to `f`. But in the case of `[1..10]`, it would be better to lose sharing to gain fusion.

This problem turned out to be central when Elliott *et al.* tried to use rewrite rules to optimise Pan programs [12]. In Pan, it is crucial to inline absolutely everything, caring nothing for sharing, apply rewrite rules, and then do aggressive common sub-expression and code-motion transformations to make up for the loss.

This is a problem that is unlikely to have a cut-and-dried solution, but we are exploring the idea of using *virtual data types*. The programmer declares some data types as *virtual*, meaning that all data structures of virtual type should be eliminated. In particular, the compiler can ignore loss of sharing when considering inlining a value of virtual type. It remains to be seen how usable such a feature would be.

5 Dynamically-generated Rules

Thus far we have concentrated on rewrite rules that are written by the programmer, but we have found that it is often useful for the compiler itself to generate rewrite rules dynamically. We give three examples in this section.

5.1 Specialisation

Haskell's type classes give rise to overloaded functions with types like this:

```
invert :: Num elt => Matrix elt -> Matrix elt
```

Such overloaded functions are somewhat inefficient: `invert` takes a tuple (or "dictionary") of functions as an extra argument, which give the arithmetic operations over values of type `elt`. Optimising compilers for Haskell allow the programmer to write a `SPECIALISE` pragma, thus:

```
{-# SPECIALISE
     invert :: Matrix Int -> Matrix Int
#-}
```

This pragma encourages the compiler to build a specialised version of `invert`, in which the matrix elements are known to be of type `Int`, giving much more efficient code. (GHC will also infer such pragmas from the types at which `invert` is called, but only within a single module.)

Suppose, then, that the compiler has constructed the specialised function, and called it (say) `invert_Int`. The next task is to make sure that suitable calls to `invert` are replaced by calls to `invert_Int`. This is where rules come in. The compiler dynamically generates a rewrite rule like this:

```
{-# RULES
 "invert/Int" forall d::Num Int.
            invert @ Int d = invert_Int
#-}
```

Unlike our earlier, programmer-specified rules, this rule is written in GHC's explicitly-typed intermediate language, called "`Core`". In `Core`, every binder has an explicit type, and polymorphism is expressed using explicit type abstraction and application. The rules written by the user in the (implicitly-typed) Haskell source code are translated into the `Core` language by the type-checker (which adds type information) followed by the desugarer (which converts Haskell's rich syntax into `Core`'s much more limited forms).

In this case `invert` is polymorphic, and so takes a type argument, indicated by the "`@ Int`" on the left hand side of the rule. It also takes an argument corresponding to the `Num elt` constraint, namely the tuple of arithmetic operations referred to earlier. So the rule simply says that a call to `invert` applied to type `Int` and tuple `d` can be rewritten to `invert_Int`. Haskell's type system ensures that there is only one possible value for the tuple of methods `d::Num Int` — namely the numeric operations on `Int` values — and these methods are "baked into" `invert_Int`, so the rule can simply discard the argument `d`.

5.2 Evaluated arguments

In array-intensive code, one often encounters a loop like this:

```
f :: Int -> Int -> Int
f x y = if x == 0 then 0
       else y + f (x-1) (y+1)
```

GHC represents values of type `Int` using the following data type:

```
data Int = I# Int#
```

where `Int#` is the type of unboxed, 32-bit integers. GHC will compile `f` thus:

```
f :: Int -> Int -> Int
f x y = case x of { I# xv -> fw xv y }

fw :: Int# -> Int -> Int
fw xv y
  = if (xv ==# 0#) then I# 0#
    else
      case y of { I# yv ->
      case fw (xv -# 1#) (I# (yv +# 1#)) of { I# rv ->
      I# (yv +# rv) }}
```

`f` has turned into a mere "wrapper" that evaluates `x` before calling the "worker", `fw` [30]. It can do this because `f` is sure to evaluate `x`. However, `f` is not certain to evaluate `y`, so the evaluation of `y` must be in the `else` branch of the conditional in the worker, `fw`. That means that the worker must re-box `y` before calling itself ("`I# (yv +# 1#)`"), and in the common case, `y` will immediately be un-boxed again. This is bad.

What can be done? Again, it is a matter of specialisation. Recognising that there is a recursive call to `fw` in which the second argument is a constructor application, GHC can make a specialised version of `fw`, and generate an appropriate rule, thus:

```
fw1 :: Int# -> Int# -> Int
fw1 xv yv = let y = I# yv
            in ...original RHS of fw....

{-# RULES "fwV"  forall xv yv.
                 fw xv (I# yv) = fw1 xv yv
#-}
```

After simplifying the right hand side of `fw1`, using the rule, we get just what we want:

```
fw1 :: Int# -> Int# -> Int
fw1 xv yv
  = if (xv ==# 0#) then I# 0#
    else
      case fw1 (xv -# 1#) (yv +# 1#) of { I# rv ->
      I# (yv +# rv) }
```

`fw` remains as an "impedance matcher" embodying the first iteration of the loop, before calling `fw1`. However the rule remains to transform any call of `f` with an already-evaluated second argument into a call to `fw1`.

All of this is done invisibly by the compiler — the programmer is not involved at all. The transformation is fully implemented in GHC, enabled by "`-O2`". The analysis, generation of specialised code, and generation of the rewrite rule, takes only 225 lines of Haskell. The rewrite-rule infrastructure automatically takes care of applying the rule when it is relevant, and propagating the rule across separate compilation boundaries.

5.3 Usage types

We are exploring another example of the same pattern. Wansbrough's work on usage types suggests that considerable efficiency gains can be made by specialising functions based on their usage patterns [38]. For example, consider `map` again:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

If `map` is called in a context in which the result list is consumed at most once, then the thunks for `f x` and `map f xs` do not need to be self-updating; instead the updates can be omitted. To express this, GHC adds extra usage-type arguments to `map`, both at its definition and at its call sites. Once this is done, a specialised version of `map` can be compiled for the case when the usage-type argument is "`once`", and a rule generated to match such calls, in exactly the same way as for specialising overloading.

5.4 Summary

In each example, we can discern the same pattern:

- Based on pragmas or program analysis, perform a local transformation (e.g., generating the specialised version of `invert`).
- Generate a rule that explains how that transformation can be useful to the rest of the program. In some cases the rule looks at the type arguments, in others at value arguments.
- Apply the rule throughout the rest of the program.

This may not sound like much, but it is extremely helpful to have a single, consistent way to propagate the benefits of a transformation to the rest of the program. For example, it is not enough for the specialiser to generate specialised versions of a function *and* find all appropriate call sites for the specialised function. There may not *be* any calls to `invert` at type `Int` when the specialiser runs. Such calls may only show up after some other inlinings have exposed them. Or they may be in other modules altogether, so the rule must be propagated across module boundaries (which is relatively easily done).

Programmer-defined `RULES` pragmas are only allowed at top level, but this is a purely syntactic restriction. Rewrite rules make perfect sense for nested functions bound by a local `let` or `letrec`, and GHC will indeed generate dynamic rules using the ideas of this section for local functions. This is important in practice, because inlining generates many nested function definitions.

6 Application: Constraint Satisfaction Problems

Next we give an example user application — solving constraint satisfaction problems (CSPs) — in which rewrite rules help support high-level, modular programming style. The added rules, which describe short-cut deforestation on rose trees, are confined to a library, and they make a representative kernel of the application run three times faster, by eliminating essentially all the overhead due to the modular style.

6.1 Modular search

Many interesting algorithms for solving CSPs are conceptually based on trees, whose nodes represent states in the search space. Solutions to the search problem are found by locating nodes that represent complete, consistent states. In a conventional imperative recursive implementation, these search trees are merely notional; they correspond to the tree of procedure activation histories. In Haskell, one can make the state tree into an explicit (lazy) data structure instead [19,5]. This approach permits search algorithms to be modularized into separate functions (really coroutines) that communicate via a lazily-constructed tree labeled with consistency information. The component functions perform generation of all possible states, consistency labeling, pruning of inconsistent states, and collection of solutions. In earlier work, Nordin and Tolmach showed that a large variety of useful algorithms — which look quite different from one another when written imperatively — can be obtained in the lazy framework just by varying the labeling and pruning functions [28].

The underlying algorithm is a simple composition of functions, where all the intermediate results are trees or lists.

```
solver :: Labeler a -> Pruner a -> CSP -> [State]
solver labeler pruner csp =
  (filter (complete csp) . map fst . leaves .
     prune pruner . (labeler csp) .
     mkSearchTree) csp
```

Here `CSP` is a type describing instances of constraint satisfaction problems; for example, we might have a function

```
queens :: Int -> CSP
```

to generate instances of the familiar n-queens problem. `State` is the type of partial solutions. Function

```
mkSearchTree :: CSP -> Tree State
```

constructs a tree of all possible partial solutions to a given CSP. Here `Tree` is the type of ordinary "rose trees," in which each node has a value and an arbitrary number of children. The `labeler` argument to `solver` has this type:

```
type Labeler a =
  CSP -> Tree State -> Tree (State, a)
```

It specifies how to attach consistency annotations to each node in the tree. The `pruner` argument, of type

```
type Pruner a = (State,a) -> Bool
```

says how to inspect the annotations to determine whether the node is consistent; `prune` removes subtrees rooted at inconsistent nodes. `leaves` returns the leaves of the tree as a list in left-to-right order. The subsequent list operations throw away the annotations and weed out nodes representing incomplete solutions.

To obtain simple back-tracking search, we can provide a `Labeler` that checks the consistency of each node individually, and annotates the node with the boolean result of the check.

```
labelInconsistencies :: CSP -> Tree State -> Tree (State,Bool)
labelInconsistencies csp = mapTree f
   where f s = (s,not (consistent csp s))

btsolver :: CSP -> [State]
btsolver = solver labelInconsistencies snd
```

More sophisticated algorithms use labelers that may look at more than one node at a time or store more information in the annotations. For example, a well-known algorithm called forward checking can be implemented by a labeler that stores a (lazily constructed) cache table of consistency information at each node.

```
labelCSCache :: CSP -> Tree State ->
                       Tree (State,Cache ConflictSet)
extractConflict :: (State,Cache ConflictSet) -> Bool

fcsolver :: CSP -> [State]
fcsolver = solver labelCSCache extractConflict
```

Interesting new combinations of algorithms can be obtained by appropriate composition of labeling functions, giving us a "mix and match" approach to algorithm construction. The modular algorithms that result are much simpler to read, write, and modify than their imperative counterparts, and have the same asymptotic behavior (in both space and time).

However, the modular Haskell code *is* much slower than equivalent C code, if only by a constant factor. We measured performance of a representative kernel of code that implements standard backtracking search on the n-queens problem and counts the number of solutions found. The modular version of this function is written

```
qsolns :: Int -> Int
qsolns n = length (btsolver (queens n))
```

On the 11-queens problem, `qsolns` runs about 30 times slower than a conventional recursive C algorithm that doesn't use trees at all. More strikingly, perhaps, it is almost four times slower than a non-modular Haskell transliteration of the C algorithm. This difference suggests that we try to fuse the tree traversals to avoid building the nodes of the several intermediate trees.

In the remainder of this section, we describe short-cut deforestation for rose trees, and discuss our experience in using rules with this application. Full code for the kernel modular code and the corresponding monolithic function are given in the Appendix.

6.2 Fusion on rose trees

We treat rose trees as an abstract data type, with public functions `initTree`, `mapTree`, `prune`, and `leaves`. The internal representation data type and `foldTree` operation are standard:

```
data Tree a = T a [Tree a]

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f t = go t
  where go (T a ts) = f a (map go ts)
```

We introduce a `buildTree` analogous to `build` on lists, and the corresponding fusion rule:

```
buildTree :: forall a.
                (forall b. (a -> [b] -> b) -> b) -> Tree a
buildTree g = g T

{-# RULES
"foldTree/buildTree"
  forall k (g::forall b.(a->[b]->b) -> b) .
    foldTree k (buildTree g) = g k
#-}
```

Now we must take care that all tree-producing functions use `buildTree`, and all tree-consuming functions use `foldTree`. Since `Tree` is as ADT, we don't need to worry about client code using the `T` constructor directly.

Function initTree generates a tree from a function that computes the children of a node [19]; mapTree is the analogue of the familiar functions on lists.

```
initTree :: (a -> [a]) -> a -> Tree a
initTree f a =  buildTree g
  where g n = go a
                where go a = n a (map go (f a))

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f t = buildTree g
  where g n = foldTree h t
                where h a ts = n (f a) ts
```

prune p t removes every subtree of t whose root value matches predicate p. Since we cannot represent empty trees, we require that p always return False on the root node of the entire tree, which is always appropriate in our applications.

```
prune :: (a -> Bool) -> Tree a -> Tree a
prune p t = buildTree g
  where
  g n = head (foldTree f t)
      where f a ts | p a = []
                   | otherwise = [n a (concat ts)]
```

Finally, leaves extracts the values at the leaves of a tree into a list in left-to-right order.

```
leaves :: Tree a -> [a]
leaves = foldTree f
  where f leaf [] = [leaf]
        f _    ts = concat ts
```

Ideally, we would like leaves to be written as a list build, so that it can fuse with list consumers further down the pipeline. Unfortunately, this seems to require doing a higher-order tree fold, which produces an intermediate list of function closures; GHC doesn't handle such lists very effectively, and it proves more efficient to stick with the simple definition shown here.

We mark all the functions to be inlined if possible.

6.3 Short-cut deforestation pays again

Given these definitions, GHC is able to completely fuse away all the rose trees in qsolns; i.e., no T constructors are applied at all! Indeed, modifying the implementation of our rose tree ADT to perform cheap deforestation improves performance of (qsolns 11) by a factor of more than three, bringing it to within 15% of the running time of a hand-fused, non-modular Haskell implementation. Moreover, this improvement comes *without requiring any changes to the search application code itself.*

All is not quite so straightforward as it may seem, however. All the problems we examined in the context of list fusion appear again for trees:

- Effective application of the fusion law requires that GHC inline more enthusiastically than it normally would. For example, our pipeline of tree operations generates many fusion opportunities that require inlining underneath the lambda of a `buildTree` argument. This is, in fact, a safe thing to do, since the lambda is "one shot," but GHC doesn't know this – and since we are thinking of trees as a user-defined library, it would be obviously inappropriate to hack this fact about `buildTree` into the compiler, the way we did for list `build`. As it happens, for the particular kernel of code we show here, GHC can use the fact that the lambda representing the entire program is one shot to deduce – after repeated iteration of inlining – that these `buildTree` lambdas are one shot as well. But in general, we need linearity analysis.
- If fusion fails, the tree library should make sure that the resulting code is not worse than it would have been had fusion never been attempted. As with lists, we must either ensure that inlining `foldTree` produces good code, or provide a "back-out" mechanism, with appropriate attention to phasing of inlining (c.f. Section 4.3).
- For full effectiveness, we need to make sure that inlining of list functions (e.g., on the lists of children in nodes) occurs only *after* inlining of tree functions (c.f. Section 4.2). We can arrange this by attaching an earlier phase number to the tree function inlining directives.
- Most seriously, we might easily write programs for which fusion fails for legitimate reasons, e.g. because there are several consumers for a given producer, or simply because we've made a mistake when writing a rule. But we'll get no feedback from the compiler about such failures. This is clearly a crucial area for further work.

7 Related Work

The basic concepts of our rules system are far from new. There have been a great many attempts to build frameworks for user-directed or application-specific optimization, often by adding additional semantic specifications to functions.

These ideas have been of particular interest in the high-performance computing community. Scientific codes often use well-established, high-level libraries, such as LINPACK or PLAPACK. Because these libraries need to work efficiently over a wide range of machine architectures and data sets, they typically have multiple implementations, each with its own complex interface. For portability and maintainability, client code should be written using portable, high-level library calls, leaving the compiler to determine the appropriate low-level calls to use and optimizing the client code accordingly. To achieve this, library interfaces can be annotated with additional specification information. Systems and proposals along these lines include TAMPR [6], Broadway [17,18], MetaScript [20], and Active Libraries [35].

Another set of systems has developed from the algebraic specification community. For example, the OPAL language [11] combines functional programming and algebraic specification in a uniform framework. OPAL laws are used to justify or guard rewrites of functional code; since laws are first-order predicate formulas over equality of functional expressions, this makes the system very powerful (and of course undecidable). It is unclear to what extent the existing implementation of OPAL supports automated optimization.

Compared to existing systems and proposals, ours is notable primarily for what it leaves out. More precisely, we can identify the following contrasts between our systems and others:

No meta language. Both left-hand and right-hand sides of our rules are just Haskell source expressions. With the exception of TAMPR [6], most of the other tools known to us operate on internal program representations, such as abstract syntax trees or control-flow graphs, and they typically allow right-hand sides to be defined using some kind of meta-programming facility. The choice of a meta-programming language is delicate. A specialized language or notation such as *metal* [13] is concise, but must be learned from scratch by the library author and can be unduly constraining; using a general-purpose programming language, such as LISP (as in early work on Aspect-Oriented Programming [21,27]) is more flexible, but requires the author to take great care to maintain essential invariants.

Simple rewrite strategy We rely on a very simple, built-in strategy, modified by "phases", for determining when and where rules should be applied. As rule sets become more elaborate, authors may need to exercise explicit control over strategy, e.g., as in Stratego [37].

Simple pattern-matching. We rely on the programmer to use high-level operators, such as `foldr`, that encapsulate control flow. Thus we don't need to provide sophisticated contextual pattern matching to identify loops or recursions, unlike systems like OPTRAN [26], Dora/Tess [15], and KHEPERA [14]. Nor do we have to deal with the unpredictability and possible high cost of higher-order matching, as used in MAG [10].

No side conditions. We work with a purely functional language, which means that many useful optimizing transformations are context-independent and don't require elaborate side-conditions. By contrast, most useful transformations on imperative programs must be justified by non-syntactic, and often non-trivial, analysis, e.g., of control flow, dependence, aliasing, etc. Thus many tools for imperative languages focus on specifying analyses in addition to transformations; examples include DFA&OPT-MetaFrame [23], Sharlit [34], Genesis [39], OPTIMIX [2], Intentional Programming [1], and recent work of Lacey and de Moor [24].

No termination guarantees; no AC rewriting. Our rules are all directed, and we cannot easily express commutative laws without causing endless rewriting. In a modern algebraic transformation system like Maude [9], equations are entirely symmetric in their left and right hand sides, which can be arbitrary terms; they can be used for transformation in either direction. Common algebraic properties of an operator can be declared by built-in keywords such as `[assoc]` and `[comm]`; in executing the transformations in a program, all pattern matching is conducted modulo these properties, which makes for shorter and more elegant programs.

In summary, we offer *simplicity* in exchange for more limited functionality. Simplicity is important, both for implementors and library authors. From an implementation point of view, our experience is that simple ideas are seldom easy to implement in a full-scale, optimising compiler, while complex ideas require heroism that is hard to sustain in the long term.

From a programming point of view, too, simplicity is important. Most particularly, the fact that the transformations are expressed entirely in Haskell itself, and not in some (necessarily different, and more indirect) meta-language is a huge advantage. We know of no optimising compiler in widespread use that supports domain-specific extensions; we suspect that this is partly due to the complexity of their meta-programming mechanisms. Of course, GHC's rules are not in widespread use by programmers either — but they are used behind the scenes in every run of GHC, both for list fusion (Section 3) and specialisation (Section 5). It is also possible that our approach is just *too* simple: we do not yet know how the tradeoff between simplicity and expressiveness will play out.

8 Conclusions and further work

We have described a simple, but fully implemented and deployed, way to write domain-specific extensions to a compiler for Haskell, by means of rewrite rules. We have demonstrated that, though simple, rewrite rules are useful in practice. Indeed, the list fusion rules have been deployed in the Prelude of the released GHC compiler for two years. In recent work, Chakravarty and Keller are using GHC's rewrite rules to perform array fusion in their work on nested data-parallel programming [7]; their application is more sophisticated than any we have described here.

The previous section described many directions in which one could imagine making our system more expressive, but we plan to develop more experience of its practical use before elaborating it much further. Indeed, the most pressing area for further work is not even mentioned in Section 7: it is the question of how best to provide feedback to the programmer about which rules have fired and, more especially, which have not and why not. Since rewrites are done on `Core`, which is quite far from Haskell, providing comprehensible feedback is a hard problem.

The status of this paper is as a report of work in progress. We present it in the hope that it will attract the interest of the writers of library packages, and will encourage them to experiment with the feature and report on its inadequacies. For the longer term, we wish to promote the principle that a programmer should supply further declarative information together with the code of the program; and suggest that compilers and other programming tools should take maximum advantage of these declarations.

Acknowledgements

We gratefully thank Manuel Chakravarty, Andy Gill, Oege de Moor, Eelco Visser, and the anonymous referees for their helpful feedback on earlier versions of this paper.

References

[1] Aitken, W., B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter and C. Simonyi, *Transformation in intentional programming*, in: *Proc. 5th International Conference on Software Reuse* (1998).

[2] Assmann, U., *How to uniformly specify program analysis and transformation with graph rewrite systems*, in: *Proc. Compiler Construction 1996*, LNCS **1060**, 1996, pp. 121–135.

[3] Baader, F. and T. Nipkow, "Term rewriting and all that," Cambridge University Press, 1999.

[4] Bird, R. and O. D. Moor, "The Algebra of Programming," Prentice Hall, 1996.

[5] Bird, R. and P. Wadler, "Introduction to Functional Programming," Prentice Hall, 1988.

[6] Boyle, J., T. Harmer and V. Winter, *The TAMPR program transformation system: Design and applications*, in: E. Arge, A. Bruaset and H. Langtangen, editors, *Modern Software Tools for Scientific Computing*, Birkhauser, 1997 .

[7] Chakravarty, M. and G. Keller, *Functional array fusion*, in: *ACM SIGPLAN International Conference on Functional Programming (ICFP'01)* (2001).

[8] Claessen, K. and J. Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, in: *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)* (2000), pp. 268–279.

[9] Clavel, M., S. Eker, P. Lincoln and J. Meseguer, *Principles of Maude*, in: J. Meseguer, editor, *Proceedings of the First International Workshop on*

Rewriting Logic, Electronic Notes in Theoretical Computer Science **4** (1996), pp. 65–89.

[10] de Moor, O. and G. Sittampalam, *Generic program transformation*, in: *Proc. 3rd International Summer School on Advanced Functional Programming*, LNCS **1608**, 1999, pp. 116–149.

[11] Didrich, K., A. Fett, C. Gerke, W. Grieskamp and P. Pepper, *OPAL: Design and Implementation of an Algebraic Programming Language*, in: J. Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, LNCS **782** (1994), pp. 228–244.

[12] Elliott, C., S. Finne and O. de Moor, *Compiling embedded languages*, in: *Proc. Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, LNCS **1924**, 2000.

[13] Engler, D., B. Chelf, A. Chou and S. Hallem, *Checking system rules using system-specific, programmer-written compiler extensions*, in: *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 2000.

[14] Faith, R. E., L. S. Nyland and J. F. Prins, *KHEPERA: A system for rapid implementation of domain specific languages*, in: *Proc. USENIX Conference on Domain-Specific Languages*, 1997, pp. 243–255.

[15] Farnum, C. D., "Pattern-Based Languages for Prototyping of Compiler Optimizers," Ph.D. thesis, University of California, Berkeley (1990), technical Report CSD-90-608.

[16] Gill, A., J. Launchbury and S. Peyton Jones, *A short cut to deforestation*, in: *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, ACM, Cophenhagen, 1993 pp. 223–232.

[17] Guyer, S. Z. and C. Lin, *An annotation language for optimizing software libraries*, in: *Proceedings of the 2nd Conference on Domain-Specific Languages* (1999), pp. 39–52.

[18] Guyer, S. Z. and C. Lin, *Optimizing the use of high performance software libraries*, in: *Proc. 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC00)*, 2000.

[19] Hughes, J., *Why functional programming matters*, The Computer Journal **32** (1989), pp. 98–107.

[20] Kennedy, K., B. Broo, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey and L. Torczon, *Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries*, Journal of Parallel and Distributed Computing (2000), (To Appear).

[21] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: *ECOOP '97 — Object-Oriented Programming 11th European Conference*, LNCS **1241**, 1997, pp. 220–242.

[22] Kirchner, C., H. Kirchner and M. Vittek, *Implementing computational systems with constraints*, in: P. Kanellakis, J.-L. Lassez and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming*, Brown University, Providence R.I., USA, 1993, pp. 166–175.

[23] Klein, M., J. Knoop, D. Koschützki and B. Steffen, *DFA and OPT-METAFrame: A tool kit for program analysis and optimization*, in: *Proc. 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, LNCS **1055**, 1996, pp. 418–421.

[24] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: *Proc. Compiler Construction 2001*, LNCS **2027**, 2001, pp. 52–68.

[25] Launchbury, J. and T. Sheard, *Warm fusion*, in: *ACM Conference on Functional Programming and Computer Architecture (FPCA'95)*, ACM, La Jolla, California, 1995 pp. 314–323.

[26] Lipps, P., U. Möncke and R. Wilhelm, *OPTRAN - a language/system for the specification of program transformations: System overview and experiences*, in: *Proc 2nd Workshop on Compiler Compilers and High Speed Compilation*, LNCS **371**, 1988, pp. 52–65.

[27] Mendhekar, A., G. Kiczales and J. Lamping, *RG: A case-study for aspect-oriented programming*, Technical Report SPL97-009, Xerox Palo Alto Research Center, Palo Alto, CA, USA (1997).

[28] Nordin, T. and A. Tolmach, *Modular lazy search for constraint satisfaction problems*, Journal of Functional Programming (2001), (To appear.).

[29] Partain, W., *The* `nofib` *benchmark suite of Haskell programs*, in: J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, Springer Verlag, 1992 pp. 195–202.

[30] Peyton Jones, S. and J. Launchbury, *Unboxed values as first class citizens*, in: *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)* (1991), pp. 636–666.

[31] Peyton Jones, S. and S. Marlow, *Secrets of the Glasgow Haskell Compiler inliner*, in: *Workshop on Implementing Declarative Languages*, Paris, France, 1999.

[32] Peyton Jones, S. and A. Santos, *A transformation-based optimiser for Haskell*, Science of Computer Programming **32** (1998), pp. 3–47.

[33] Reid, A., *The Hugs graphics library*, Technical report, School of Computing, University of Utah (2000).

[34] Tjiang, S. and J. Hennessy, *Sharlit – a tool for building optimizers*, in: *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, 1992, pp. 82–93.

[35] Veldhuizen, T. L. and D. Gannon, *Active libraries: Rethinking the roles of compilers and libraries*, in: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)* (1998).

[36] Visser, E., *Strategic pattern matching*, in: *Rewriting Techniques and Applications (RTA'99), Trento*, Lecture Notes in Computer Science (1999).

[37] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *Proceedings of the International Conference on Functional Programming (ICFP'98)*, 1998, pp. 13–26.

[38] Wansbrough, K. and S. Peyton Jones, *Once upon a polymorphic type*, in: *26th ACM Symposium on Principles of Programming Languages (POPL'99)* (1999), pp. 15–28.

[39] Whitfield, D. and M. L. Soffa, *The design and implementation of Genesis*, Software — Practice and Experience **24** (1994), pp. 307–325.

Appendix: Constraint Satisfaction Problems

Here is the complete code for the constraint satisfaction problem (CSP) search kernel described in Section 6

Problem Definition

A CSP is characterized by a number of variables `vars`, a number of values `vals`, and a consistency relation `rel` between pairs of assignments of values to vars. We represent assignments using an infix constructor `:=`. To solve the CSP, we must assign a value to each variable such that all pairwise combinations of assignments are in `rel`. A well-known example is the n-queens problem, under the standard optimization that we only try to place one queen in each column; this can be modeled as a CSP with n variables (the columns), n values (the rows), and a relation that permits two assignments provided the corresponding positions are on different rows or different diagonals.

```
type Var = Int
type Value = Int

data Assignment = Var := Value

type Relation = Assignment -> Assignment -> Bool

data CSP = C {vars, vals :: Int, rel :: Relation}
```

```
queens :: Int -> CSP
queens n = C{vals=n,vars=n,rel=safe}
  where safe (col1 := row1) (col2 := row2) =
          (row1 /= row2) &&
           abs (col1 - col2) /= abs (row1 - row2)
```

Search States

We model each state in the space of possible solutions as a sequence of assignments, together with the number of the most recently assigned variable. States are built from `emptyState` by repeated use of `extensions`, which takes a state and constructs a list of extended states formed by assigning each possible value to the next variable.

```
data State = S [Assignment] Var

emptyState :: CSP -> State
emptyState C{vars=vars} = S [] 0

extensions :: CSP -> State -> [State]
extensions C{vars=vars,vals=vals} (S as lastvar) =
  [S ((nextvar := val):as) nextvar |
    let nextvar = lastvar+1, nextvar <= vars, val <- [1..vals]]

complete :: CSP -> State -> Bool
complete C{vars=vars} (S _ lastvar) = lastvar == vars

consistent :: CSP -> State -> Bool
consistent _           (S []     _) = True
consistent C{rel=rel} (S (a:as) _) = all (rel a) as
```

A solution is a complete state that is consistent at every level.

Rose Trees

Here is sample library code for rose trees written without concern for fusion. For convenience, we do use `foldTree` in the definition of `prune` and `leaves`.

```
data Tree a = T a [Tree a]

initTree :: (a -> [a]) -> a -> Tree a
initTree f a = go a
  where go a = T a (map go (f a))

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f t = go t
```

```
  where go (T a ts) = f a (map go ts)

mapTree  :: (a -> b) -> Tree a -> Tree b
mapTree f (T a ts) = T (f a) (map (mapTree f) ts)

prune :: (a -> Bool) -> Tree a -> Tree a
prune p t =
    head (foldTree f t)
      where f a ts | p a = []
                   | otherwise = [T a (concat ts)]

leaves :: Tree a -> [a]
leaves = foldTree f
  where f leaf [] = [leaf]
        f _    ts = concat ts
```

Rose trees supporting fusion

The code for these was shown in Section 6.2 .

Backtracking Search for CSPs

```
mkSearchTree :: CSP -> Tree State
mkSearchTree csp = initTree (extensions csp) (emptyState csp)

type Labeler a = CSP -> Tree State -> Tree (State, a)

type Pruner a = (State,a) -> Bool

labelInconsistencies :: Labeler Bool
labelInconsistencies csp = mapTree f
  where f s = (s,not (consistent csp s))

solver :: Labeler a -> Pruner a -> CSP -> [State]
solver labeler pruner csp =
  (filter (complete csp) . map fst . leaves .
     prune pruner . (labeler csp) .
     mkSearchTree) csp

btsolver :: CSP -> [State]
btsolver = solver labelInconsistencies snd

qsolns :: Int -> Int
qsolns n = length (btsolver (queens n))
```

Hand-fused Code

A hand-fused version of `qsolns` in Haskell:

```
qsolns' :: Int -> Int
qsolns' n = f (emptyState csp)
  where
    csp = queens n
    f state | complete csp state = 1
            | otherwise = g (extensions csp state)
    g [] = 0
    g (s':rest) | consistent csp s' = f s' + g rest
    g (_:rest) = g rest
```